

# Machine Learning Engineer Nanodegree

---

## Capstone Project

Charlie Garavaglia

March 5, 2018

## I. Definition

### Project Overview

This project constitutes an entry in a Kaggle competition, the [Statoil/C-CORE Iceberg Classifier Challenge](#).

From the description:

Drifting icebergs present threats to navigation and activities in areas such as offshore of the East Coast of Canada.

Currently, many institutions and companies use aerial reconnaissance and shore-based support to monitor environmental conditions and assess risks from icebergs. However, in remote areas with particularly harsh weather, these methods are not feasible, and the only viable monitoring option is via satellite.

Statoil, an international energy company operating worldwide, has worked closely with companies like C-CORE. C-CORE have been using satellite data for over 30 years and have built a computer vision based surveillance system. To keep operations safe and efficient, Statoil is interested in getting a fresh new perspective on how to use machine learning (ML) to more accurately detect and discriminate against threatening icebergs as early as possible.

In this competition, you're challenged to build an algorithm that automatically identifies if a remotely sensed target is a ship or iceberg. Improvements made will help drive the costs down for maintaining safe working conditions.

Much like the "blips" on ships' radar in the movies, objects in the ocean are returned to the Sentinel-1 satellite as a bright spot on an image from its Synthetic Aperture Radar (SAR). The satellite transmits a radar pulse and then records the echo of that radar bouncing off solid objects, which reflect back the energy emitted by Sentinel-1 more strongly than its surroundings (since in technical language, solid objects have a higher *reflectance*). The radar energy collected by the satellite is referred to as *backscatter* and its intensity makes up the radar images presented herein. Additionally, this particular satellite is equipped with "side looking radar", and thus views its areas of interest at an angle, rather than directly above. This angle, measured perpendicularly from the earth's surface to the area of interest, is called the *incidence angle* of the satellite. It is noted in the competition background information that "generally, the ocean background will be darker at a higher incidence angle". Finally, Sentinel-1 can transmit and receive radar energy in different planes. The combination of transmission plane and receiving plane is called the *polarization* of the radar band.

After identification on the SAR image described above, analysis is required to correctly identify the remotely sensed object. In our case, we are given the task of discriminating between ocean-going vessels and floating icebergs, which pose a significant danger. A [recent article from the European Space Agency](#) exposes the

apparent benefits from such technology: the ability to programmatically detect threats to shipping and "navigate through...notoriously icy waters". The approach presented in this paper will leverage deep learning and convolutional neural networks (CNNs) to learn what which features of a radar image determine an iceberg. A similar approach applied to the same problem is outlined by [Bentes, Frost, Velotto, and Tings \(2016\)](#), going as far to also be written in Python, but using the ML library Theano. The solution presented here will differ by preprocessing methods, the extra feature of the incidence angle, the software backend supporting the neural network (TensorFlow via Keras), and the specific CNN architecture employed to perform the classification.

My prediction is that best results will come with a custom architecture using the Bentes normalization, since we don't have access to his architecture but his methods seem sound.

## Input Data

To help accomplish our task, Statoil and C-CORE have provided us with two `json` files, one of labeled training examples (`train.json`) and one of unlabeled examples to test our model on (`test.json`). Each entry in both essentially contains a 75x75 pixel image composed of the backscatter levels from two different polarizations, along with the incidence angle the radar was emitted/collected at, and a label, if applicable.

## Problem Statement

The aim of the competition is to distinguish an iceberg from a ship in satellite radar data, specifically images from the Sentinel-1 satellite constellation's C-Band radar. Each image provided contains only an iceberg or an ocean-going ship. Thus, labeling each entry in the test set as 1 for iceberg and 0 for ship (or not-iceberg) is perfectly reasonable, as is producing a probability that the image contains an iceberg.

Thus, overall, we may view our task as a binary classification problem: does an image contain an iceberg or not?

## Solution Strategy

Employing computer vision to detect if an object in an image is a well-supported approach to this problem. This can be accomplished by building a CNN and training it on the data to recognize features of our label, in our case an iceberg. See [Hasanpour et al \(2018\)](#), [Springenberg, Dosovitskiy, et al \(2015\)](#) and others for numerous applications of CNNs for object detection. Once trained, a probability that a radar image contains an iceberg can be outputted.

## Metrics

The official evaluation metric of this Kaggle competition is the logarithmic loss, or "log loss", of our predictions. Also referred to by "binary cross-entropy" in ML, this is a suitable metric, as it gives a notion of distance from our outputted probabilities (a number between 0 and 1) to the actual label, which is either a 0 or a 1.

Detailed explanations of log loss are [available online](#), but the formula for log loss is presented here:

$$-\frac{1}{N} \sum_{i=1}^N y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

where  $y_i$  is the label (1 = iceberg, 0 = no iceberg) for image  $i$ ,  $p_i$  is the probability outputted by our model that image  $i$  contains an iceberg, and  $N$  is the total number of images.

The individual log loss is summed and divided by  $-N$  to give a comparable metric between predictions of different size and in order to make the log loss positive to preserve the notion of minimizing loss.

For a single image, if our prediction matches the label, a confident prediction probability will contribute little to the total log loss. However, if our prediction does not match, a confident incorrect prediction will contribute heavily to the log loss. We shall endeavor to minimize the log loss; a perfect classifier (that outputs 1.0 for all iceberg images and 0.0 for all non-iceberg images) would have a log loss equal to zero.

In practice, functions that calculate log loss do not directly compute predictions of exactly 1 or 0. You might notice that a perfectly wrong prediction, such as predicting a definite in image  $j$  containing a ship ( $y_j = 1$ ) with probability  $p_j = 0$  involves the calculation

$$y_j \log p_j = 1 \log 0 = \inf$$

Since we can't sum infinity meaningfully, the log loss function assigns a minimum and maximum probability that it sets predictions of 0.0 and 1.0 to, respectively.

## II. Analysis

### Data Exploration

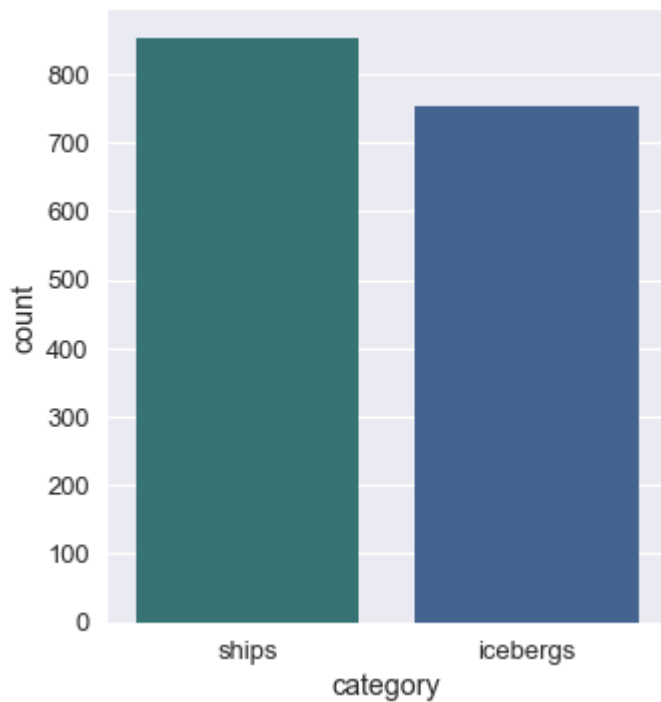
The dataset is provided by Statoil, an international energy company, C-Core, who built the satellite computer vision system we are endeavoring to improve upon, with delivery provided by the good people at [Kaggle](#).

The two `json` files, named `train.json` and `test.json`, divide the data into training and testing sets respectively. They are identical except the latter does not have a `is_iceberg` column and is much larger than the former.

For completeness, an entry in `train.json` and `test.json` contains the following fields:

- `id` = ID of the image, for matching purposes
- `band_1`, `band_2` = list of flattened image data in two bands. Each band is a 75x75 pixel image, so each list has 5625 elements. Values are decibels (dB) of radar backscatter at a given incidence angle and polarization. The polarization of `band_1` is HH, where the radar is transmitted and received in the horizontal plane. The polarization of `band_2` is HV, i.e. transmitted horizontally and received vertically.
- `inc_angle` = incidence angle of radar image. Some entries have missing data and are marked as `na`.
- `is_iceberg` = target variable: 1 if iceberg, 0 if ship. Again, this field exists only in `train.json`

There are 851 (53.05%) training examples of ships and 753 (46.95%) training examples of icebergs.

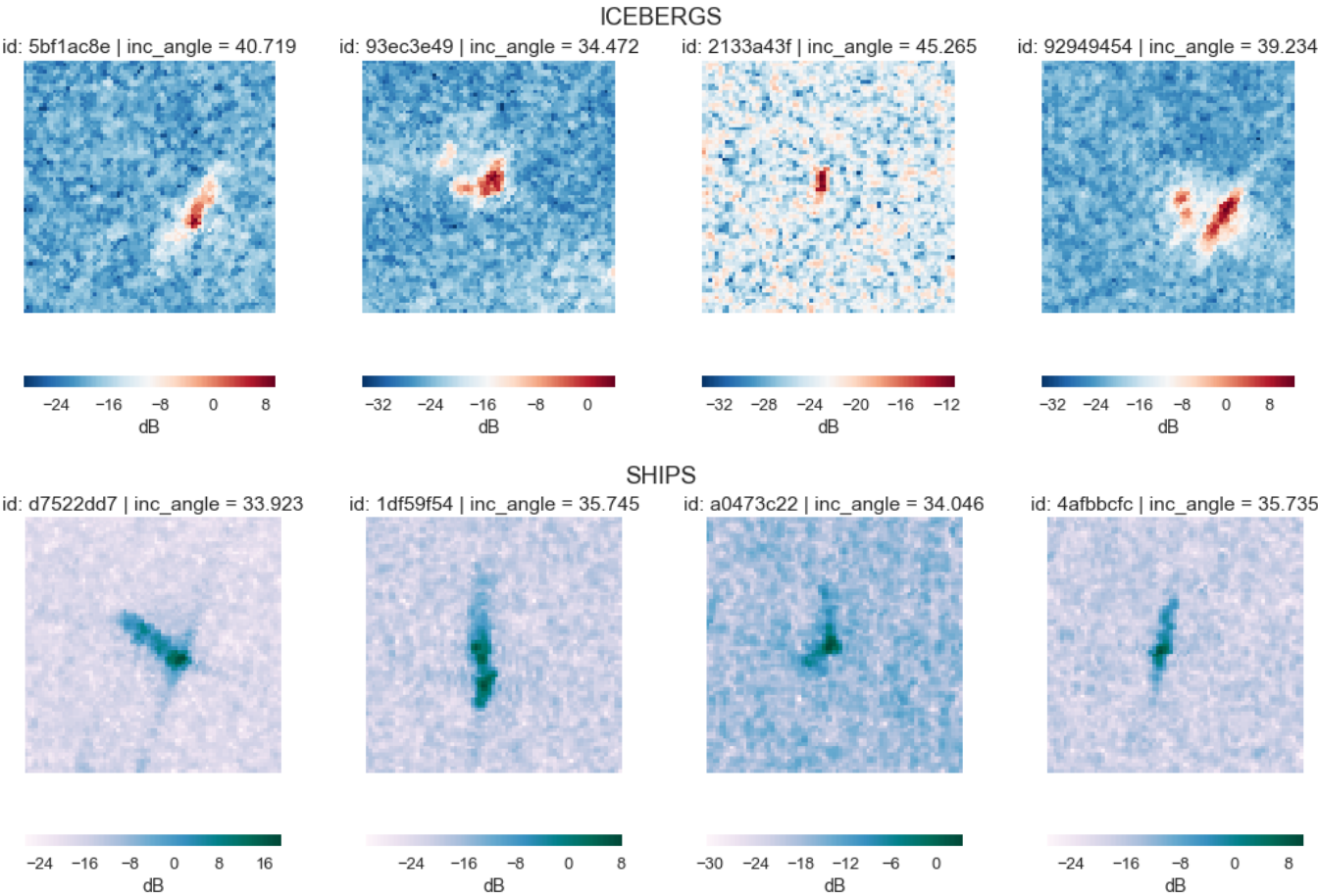


This means that (1) we must create our own validation set from the smallish dataset, and since this doesn't leave us with a lot of training examples, we must therefore (2) find ways to augment the training set with `keras` methods to improve the model's predictive power.

### **Radar Data**

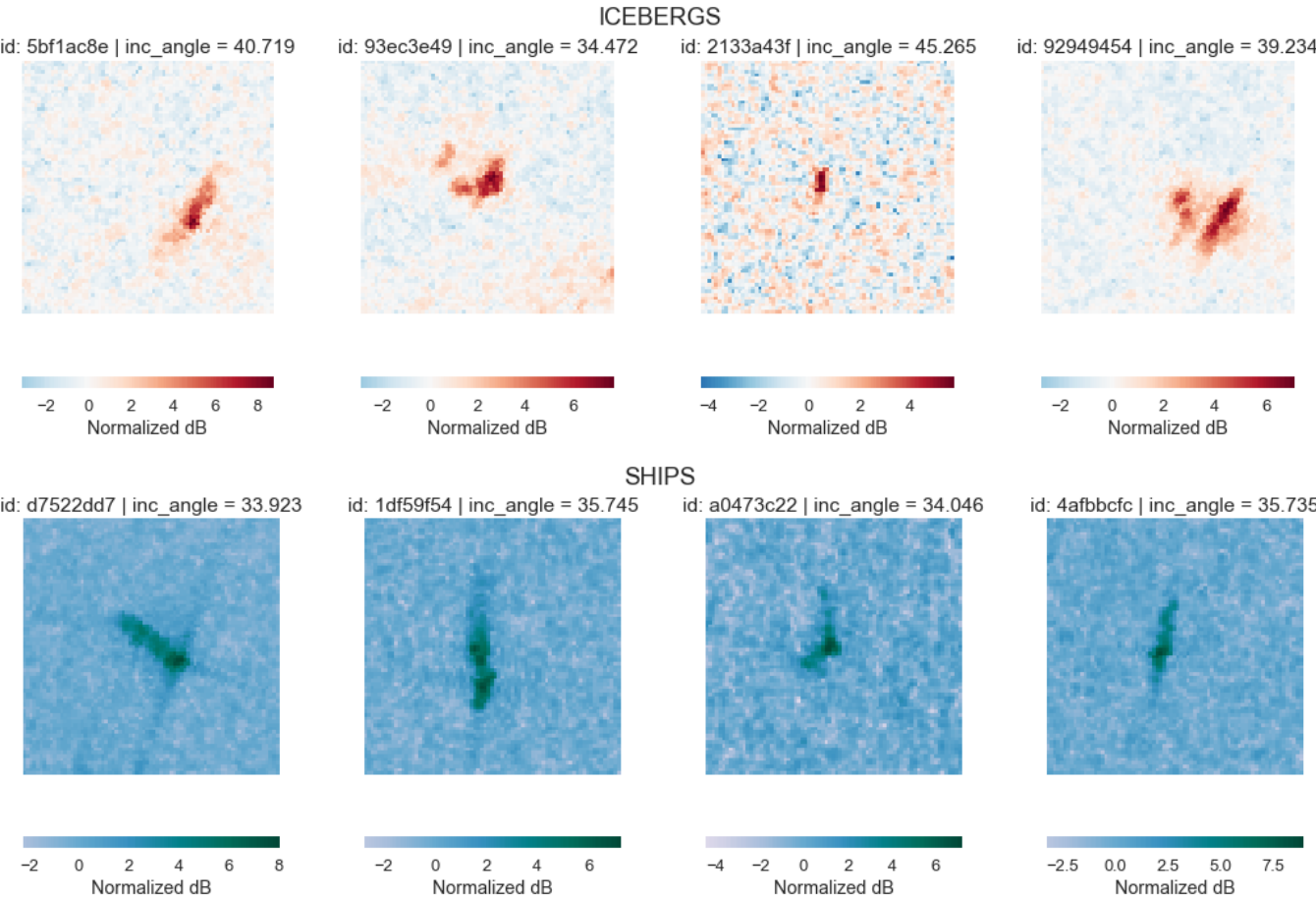
As noted in the introduction, a radar band's image data is composed of the decibel levels of backscatter, equivalent to the intensity of the reflected radar pulse for some predetermined pixel-to-area conversion. See below for heatmapped 2D illustrations:

#### **Before By-Image Normalization**



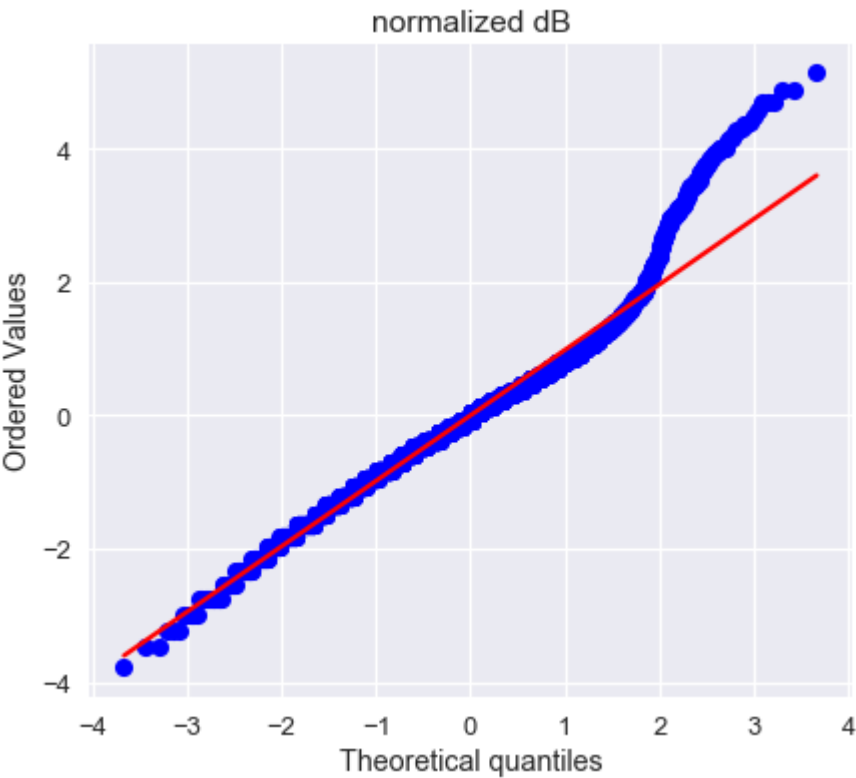
The background is somewhat muted and the area of interest stands out with bright intensity. While this is good for human classification, our model will expect normalized data.

After By-Image Normalization



All above are examples of the `band_1` aka transmit/receive horizontally (HH) radar data. We choose one of the icebergs and display its Q-Q plot, which measures the data's normal distribution tendencies.

Normalization by Image



Note that it is somewhat normally distributed, but the bright spots of the area of interest (i.e. the iceberg or ocean-going vessel) makes it more heavily right-tailed. For our first pass, we are going to leave the normalized values where they are and observe performance.

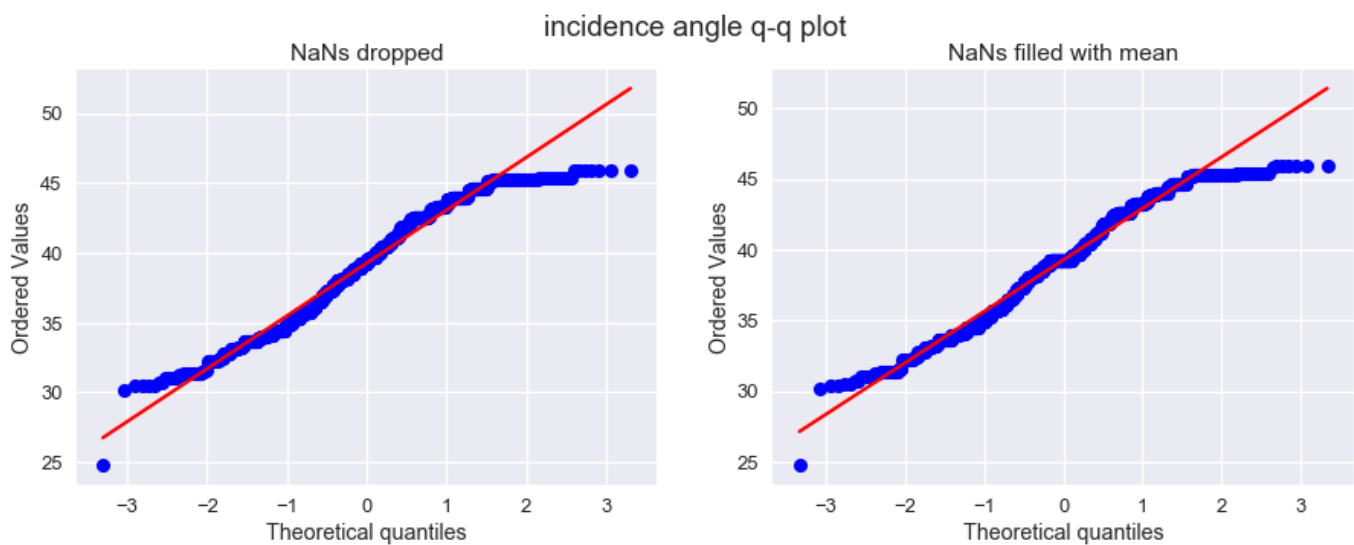
Also to note is that we will be normalizing over each band, rather than by pixel across the entire dataset. This is for the now obvious reason that the backscatter from the icebergs' and ships' areas of interest are not fixed (since icebergs are, in fact, not uniformly made). Therefore the model should learn to expect edges in different places in the image.

Finally, we must add a third channel set to the mean of `band_1` and `band_2`, since Keras expects 1- or 3-channel images (equivalent to grayscale or RGB images respectively). This is accomplished through the `process_df` function applied early in the Jupyter notebook.

## Incidence Angle

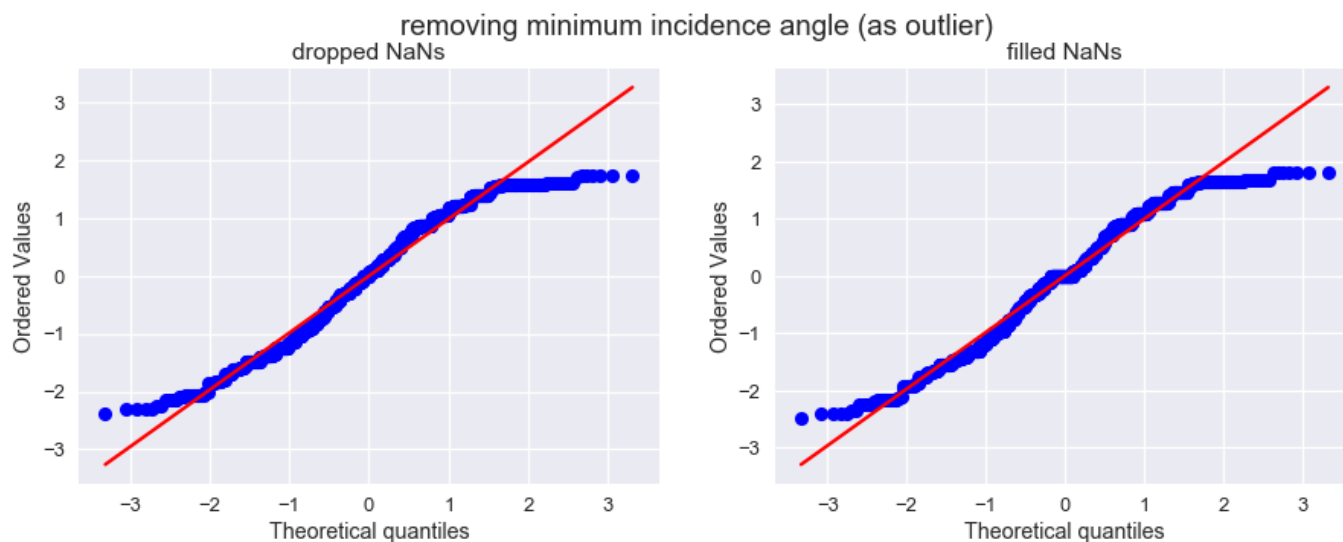
Since this is side-looking radar from a satellite, backscatter levels can be affected by the viewing and receiving angle, which may be important to include in our model's analyses.

Kaggle alerts us to the fact that the training data involves a number of missing incidence angles, all occurring when the image contains an ocean-going vessel. See below.

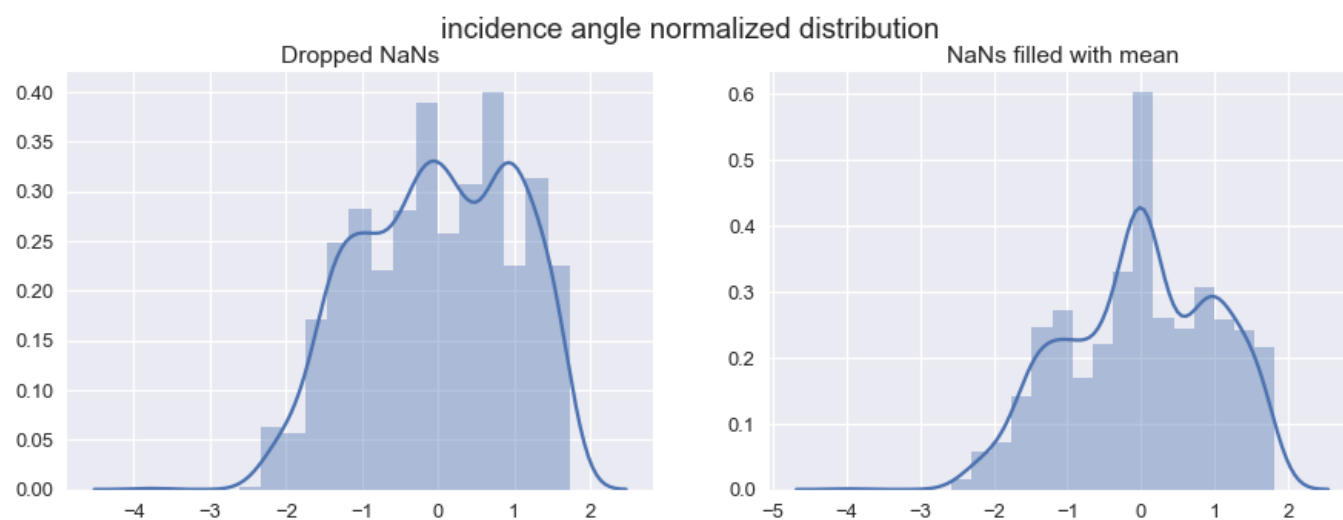


After classifying them as `np.NaN`, a decision must be made on how to handle the missing data. At this juncture, filling the missing incidence angles with the mean of incidence angles from ships only seems like a reasonable first step.

Next, a minimum value is well away from the normalized values when viewed on a Q-Q plot.



I chose to drop this minimum incidence angle as an outlier and fill the missing angles, since the training set is on the smaller size anyway. You can see the effects on the following Seaborn distplot:



What we are left with is a somewhat normal-ish looking distribution which hopefully strikes the balance between lack of data and working within Keras limitations.

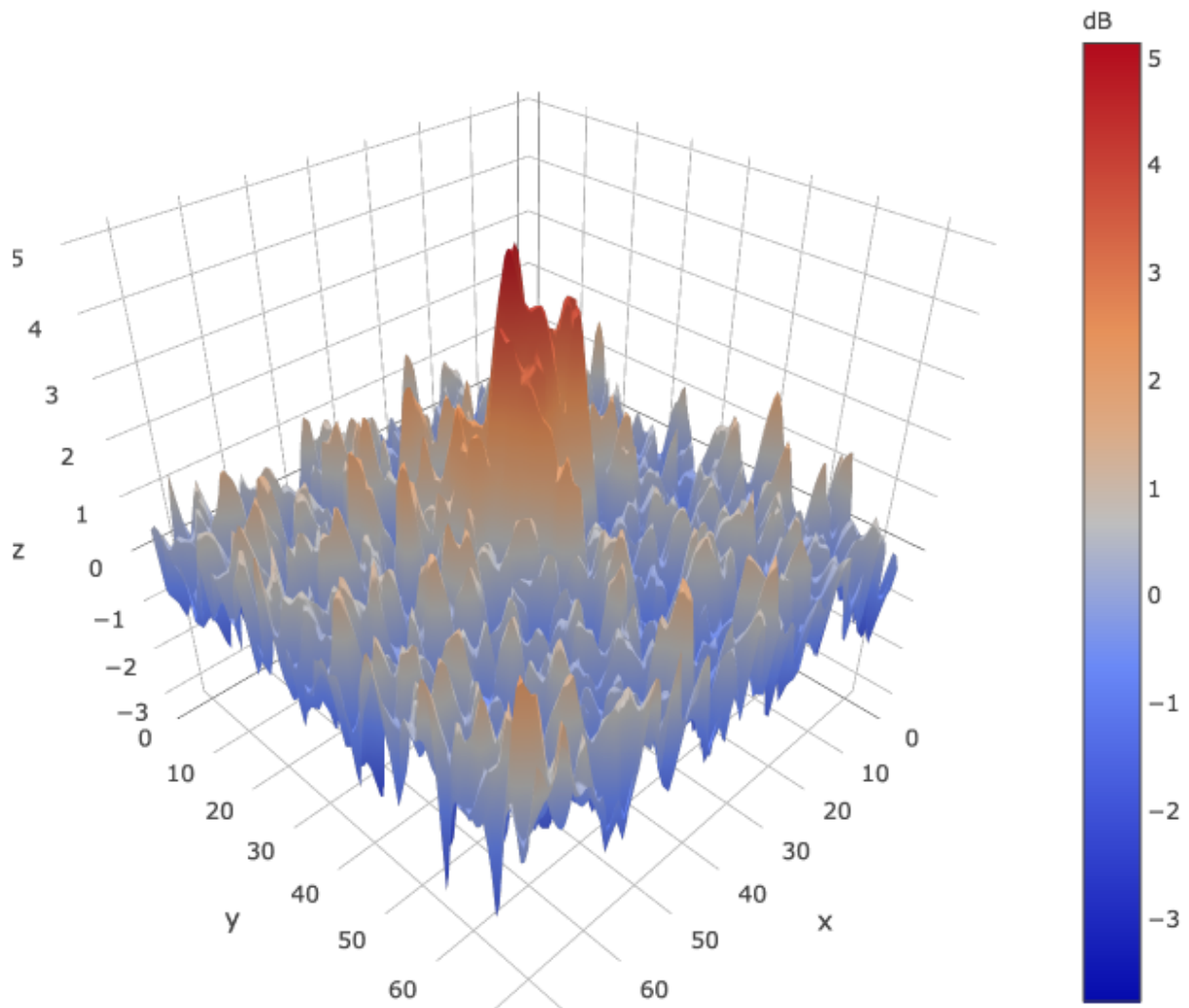
## Exploratory Visualization

I invite you to explore an interactive surface plot of an iceberg or ship thanks to [plotly](#) and [this helpful Kaggle notebook](#).

Interactive features are located in the [project report notebook](#), but below is a static view.



iceberg\_f67babb0 | normalized dB



## Algorithms and Techniques

The approach employed will largely mirror cited articles. After pre-processing and normalization techniques are applied, a convolutional neural network, or CNN, will be constructed and trained on the labeled radar images. This is the computer vision industry standard approach to object recognition in images.

Generally, a CNN works by creating some combination of layers specific to CNNs.

The first and most important for detecting edges and shapes are the convolutional layer. This is usually a square filter slid across and down the image radar images that learns the shapes associated with that particular label (i.e. 0 for ship and 1 for iceberg). This deepens the spatial information available on a particular pixel. The second notable layer is a pooling layer, which averages or maxes a subset of usually  $\geq 2 \times 2$  pixels on the radar image. This layer of the CNN usually follows a convolutional layer and works to shrink the dimensions of the informational object, while keeping the deepness that the convolution added. Finally, the other layer to be conversant in is the dropout layer, which randomly turns off some neurons in the network so

that all of the architecture can be trained. This can be thought as "turning off" your dominant hand to learn how to sign your name with your off-hand, but in the case of nodes in a network. The theory is that, unlike a person, a fully-utilized neural network will generalize better than one with "asleep" nodes.

SimpNet, like most CNN architectures, works because it does, rather than some application of theory. One of my custom CNN builds scored better on this dataset, but that code was lost in a freak git accident. SimpNet was therefore found for its accuracy on many standard datasets, such as CIFAR10, CIFAR100, MNIST, and SVHN, and its block-like design, allowing for possible refinement later.

In the paper, they reach for a combination of training simplicity and parameter reduction with performance. In their words, "it is highly desirable to design efficient (less complex) architectures with smaller number of layers and parameters, which are as good as their deeper and more complex counterparts. Such architectures can then be further tweaked using novel tricks from the literature" ([Hasanpour et al, 2018](#)).

Also to note is that we haven't mentioned the incidence angles yet. If we went through the trouble of filling the missing angles, we might as well use them in our initial analysis, right?

To address this, we must use the functional model of Keras to combine the output of the CNN with the scaled incidence angle using a `concatenate` layer. This allows multiple inputs to be combined in a single layer, which will then be fed forward to the fully connected layers for further analysis.

## Benchmark

To benchmark our solutions locally, we rely on the performance of a "vanilla" neural network. This non-CNN has a few layers of fully-connect nodes and outputs the same probability values that our CNN will. By uploading to Kaggle and viewing its log loss, we obtain a baseline from which to iterate upon.

Furthermore, baseline model results have been calculated from setting outputted probabilities to specific values:

We know a perfect classifier would have a log loss of exactly zero, but we don't know how an unintelligent classifier would fare. Three such classifiers jump to mind: one that always classifies an image as containing an iceberg at probability 1.0, one that always outputs iceberg probability 0.5, and one that's certain there's never an iceberg anywhere (iceberg probability = 0.0). An additional classifier is one that predicts an iceberg in an image equal to the frequency of icebergs in the training set

We must submit each to the [Kaggle submission page](#) for calculation of log loss, as we do not have access to the true labels of the testing data.

We have since taken as gospel the fact that the leaderboard is calculated with the private score which is 80% of the data and combined it with the public score which is calculated with 20% of the data. Even though it's not explicit that the intersection of these two subsets is non-empty, we pretend it is.

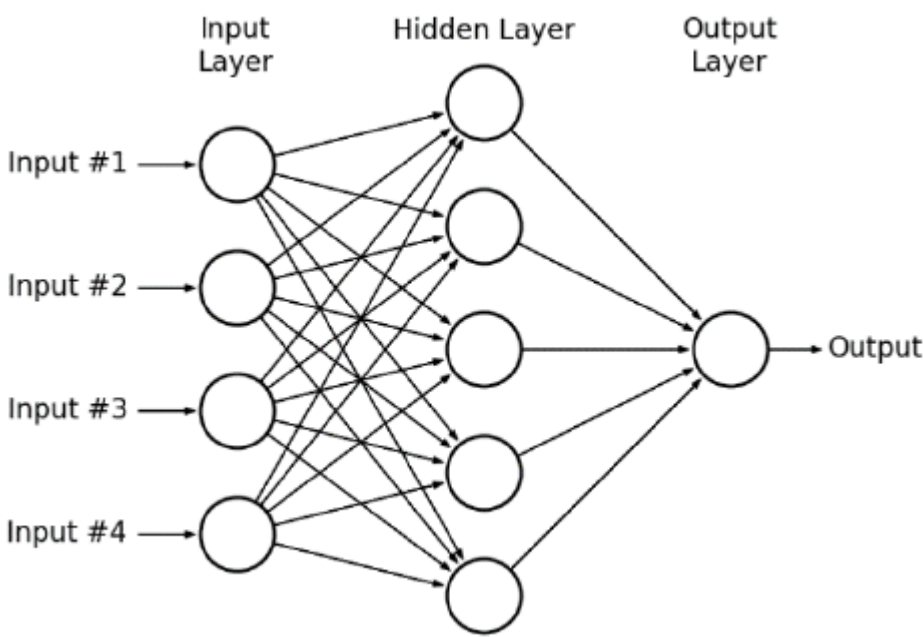
Baseline Model	<code>is_iceberg</code> =	Log Loss
certain iceberg	1.0	16.38164
certain not iceberg	0.0	18.15736
indecisive	0.5	0.6931
proportion of icebergs in training	0.469451	0.69808

As a comparison:

Model	Log Loss (private 80%)
leader	0.0822

The **certain iceberg** and **certain not iceberg** baseline models can be thought of as the worst we can do, without having access to the labels and guessing completely wrong each time. We can also infer that there are more ship examples than icebergs in the testing set, as confirmed by a constant prediction of **0.469** being slightly better than a constant guess of **0.5**.

Another benchmark is how a "vanilla" neural network would perform. By vanilla, we essentially mean a **multilayer perception**, with 3 fully connected layers of fully-connected does: one layer is the input, one the hidden, and one is the output, which consists of a single node. A picture is instructive in this case:



Our vanilla or naive neural network performed somewhat better than the constant-outputting baseline models. We endeavor to outperform this benchmark.

Baseline Model	Log Loss
Naive Neural Network	0.4553

### III. Methodology

#### Data Preprocessing

As stated, after importing into a **pandas.DataFrame**, each of the two bands are flattened Python lists of floats.

Keras expects the tensors (aka the packaged 3-banded radar image) to have the following dimensions:

```
(number of samples, height, width, 3) = (num_samples, 75, 75, 3)
```

Each pixel in the 75x75 image contains a vector with 3 entries for 3 bands. Two functions act to create a Keras 4D tensor, `helpers.process_df` and `helpers.make_tensors` (latter in the project notebook as well).

### `process_df(df)`

Given a DataFrame, this function adds the mean of the first two bands as the third band in a new column, and also reclassifies the incidence angles to `float64` with NaNs.

The `if not isinstance` statement makes sure we're not reclassifying an array that has already been reclassified. Also, we mute `FutureWarnings` related to internal Pandas improvements when changing the incidence angle to floats by the `with warnings.catch_warnings():` and the simple warnings filter.

### `make_tensors(df)`

This function creates a package of 4-dimensional tensors for Keras in a couple of steps:

1. Convert each band to a DataFrame-long `np.array`
2. Scale/normalize each array in the long `np.array`
  - Since target object is not fixed, normalizing over the image is appropriate so that the CNN gets data it likes
3. Stack each band's flat arrays on top of each other
  - Now we have 3 bands of radar data, each of the same  $75 * 75 = 5625$  element long array
4. Stack each normalized band along the last axis so that every pixel in the flattened array has 3 values, one for each band
5. Finally, reshape each array into a 75x75 pixel image with 3 bands at each pixel

We selected all features at this stage and let the nodes be active or quiet depending training.

### `inc_angle`

Further improvements might refine this section using a published transformation method.

Finally, the incidence angle has been preprocessed by filling NaNs by the mean and dropping the entry with the lowest angle as noted above.

## Implementation

### Augmentation

As always, the first thing we do is separate our training data into training and validation subsets. With a validation size of 12.5% of the training DataFrame, we have 1402 training tensors and 201 validation tensors out of 1604 total tensors. Testing data will be loaded later and scored by Kaggle.

By visualizing 4 random ships and icebergs, we can detect the general characteristics allow us as humans to discriminate ships and icebergs (ships having regular/straight edges with sometimes a haloed effect, but the CNN will zero in on to pick apart the harder cases.

I set the batch size to something manageable and a multiple of my processors, for my own comfort and to take advantage of parallel processing. Then, a Keras image data generator was created that will allow us to augment the training set by randomly creating slightly different icebergs or ships each time allowing for

multiple passes over the training set. I chose to employ horizontal and vertical flips, shifting the image randomly up to 10% horizontally or vertically, rotation up to 45 degrees, zooming up to 10%, and filling any gaps by wrapping the edges, which should be waves.

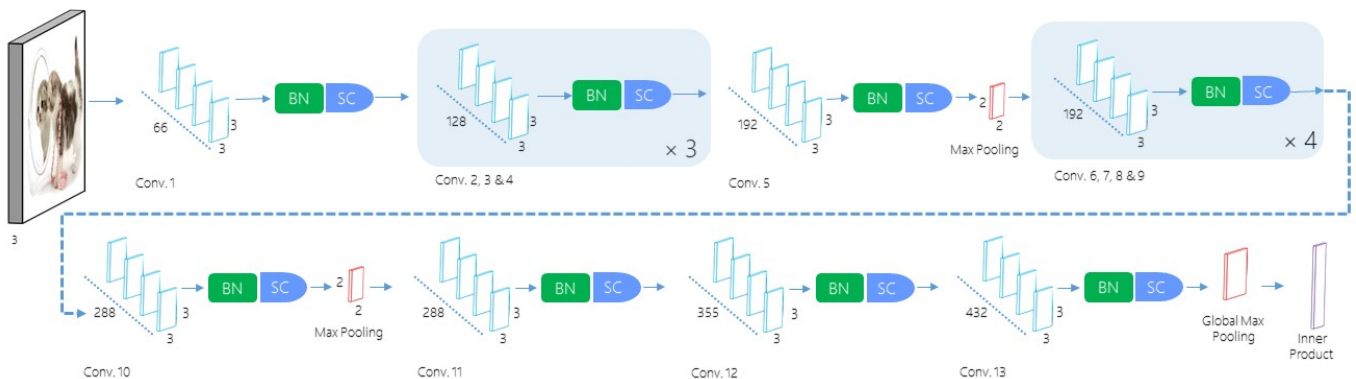
The next section expands this augmentation functionality to handle the auxiliary input of the incidence angle. Supposing we want to shuffle the training tensors, it flattens each image into a flat tensor with  $75 \times 75 \times 3 = 16875$  elements and adds the angle and target column to create a big matrix called  $Z$ . We shuffle  $Z$ , then strip off the angles and targets to use in Keras data generator. To ensure we're looping around the shuffled angles the same way as the shuffled tensors, we must use an `itertools.cycle`. Finally, the generator that outputs the 2 inputs for our is created. The `next(img_gen)` takes a certain amount of images from shuffle, thus we use the `itertools` take recipe to take only a certain amount of matching incidence angles. A similar process happens for a testing generator.

Since we wanted to randomize not only the tensors and targets, which keras supports with their `datagen.flow` functionality, but also the incidence angles, this custom generator was necessary.

Finally, we actually create the training and validation generators named `train_flow` and `valid_flow`.

## Model Building

The model itself is based on SimpNet architecture advanced by [Hasanpour, Rouhani, et al \(2018\)](#). Their design scores highly on standard datasets such as CIFAR10, CIFAR11, MNIST, and SVHN, but it is unclear whether this generalizes to efficiency with this problem domain. The selection of SimpNet was motivated by a desire to sidestep the need to create another custom architecture for the problem after a git mishap destroyed the well-performing CNN code. While it is somewhat as effective, we will leverage its block-like design and relatively low complexity to iterate on its performance.



Above is the copied design. They actually specify a whole procedure for gradually expanding the network to the classification problem, so we are operating sub-optimally by co-opting their architecture whole. I argue this will be addressed in the refinement section and there will be a demonstrated improvement over the baseline model.

The CNN ends up having 13 convolutional layers interspersed with max pooling layers and something called "SAF-Pooling" which are "essentially the max-pooling operation carried out before a dropout operation" (Hasanpour et al, 2018). Each convolutional block has a batch normalization and a "scaling" (interpreted to mean an activation function) following it. Every 5 convolutions a max pooling layer is inserted, which we took to mean the SAF-pooling operation mentioned earlier. At present, these layers are the only place dropout occurs in the CNN, at a rate of 50%. Setting convolutional blocks to have dropout layers had no or negative effects on performance.

The next code block specifies some Keras callbacks, which help us influence the training process. Specifically, we create four callbacks:

1. **ModelCheckpoint** = saves the model with the best weights according to validation. This code also creates a separate directory in case it isn't already there.
2. **EarlyStopping** = stops training early once it becomes clear that we are overtraining the model, usually used in conjunction with the above; here patience (the number of epochs until termination) is set to 10 because epochs are short on my computer
3. **ReduceLROnPlateau** = reduces learning rate when validation loss doesn't improve for a certain amount of epochs. From the docs: "Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced." Thus learning becomes more coarse and easier to generalize, or at least that is the theory
4. **TQDMNotebookCallback** = tqdm makes pretty epoch progress bars for Keras model training in Jupyter notebook which I personally enjoyed. These may have to be removed because on second viewing the Jupyter object loses its state and the pretty effect is lost.

We create these callbacks and convert them to list for compilation keyword arguments. Our **loss** to monitor is logarithmic loss, termed here as **binary\_crossentropy**. Adam is chosen as the optimizer since this is "appropriate for not-stationary objectives and problems with very noisy and/or sparse gradients," and because it typically "requires very little tuning" (Kingma & Ba, 2017). Finally, **accuracy** was chosen as a metric to supplement validation log loss as an intuitive measure of performance.

## Model Creation and Running

Since we've packaged our model compilation keywords into a dictionary, we can unpack them using the **\*\*** operator into keyword arguments. We do so and are met with the (supposedly) simple CNN that is SimpNet.

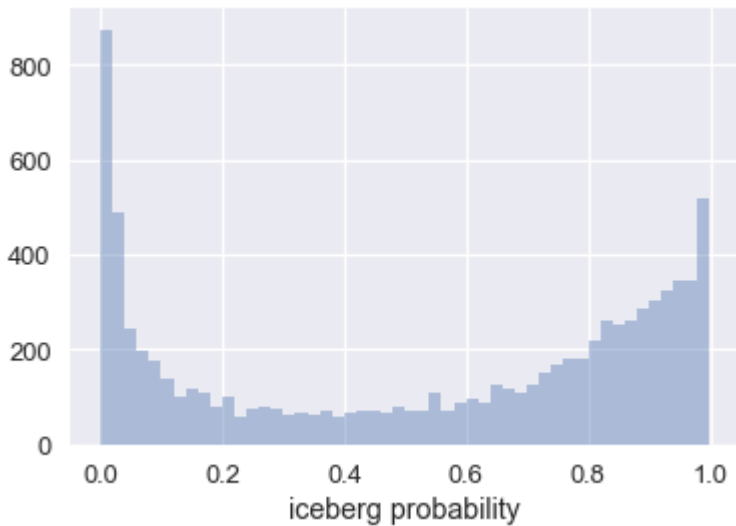
Next, we set an epoch maximum (which will never be attained due to early stopping), a desired training size (we will augment the training set a few times over to get there), and a reasonable steps per epoch (defined as the number of batches of training samples per epoch).

With all the necessary inputs to the model accounted for, we call **.fit\_generator** to fit batches of augmented images via **train\_flow** so that our model starts to discriminate between icebergs and not-icebergs. Each pass the weights get updated according to the closeness of the guess towards the true label. Validation is performed each epoch using **valid\_flow**, a single image generator that is necessary when using multiple inputs to the validation. A surprising fact may be that I originally set verbose to 0, since the tqdm widget was so cool. This might have to be reverted so you can see the results of training.

In all, 25 epochs were required to reach a minimum validation loss. Note that one reduction of learning rate allowed for additional gains in validation loss. The training was stopped after the 34th epoch and the best weights from the 25th epoch was saved.

## Prediction and Submission

All that's left is to run the predictions on the testing data and submit it to Kaggle for scoring.



I am heartened by the predictions distribution; it seems that our model is becoming confident in its predictions as evident by the many probabilities near 0 and 1. We'll see if that is a good or bad thing when we submit to Kaggle.

## Refinement

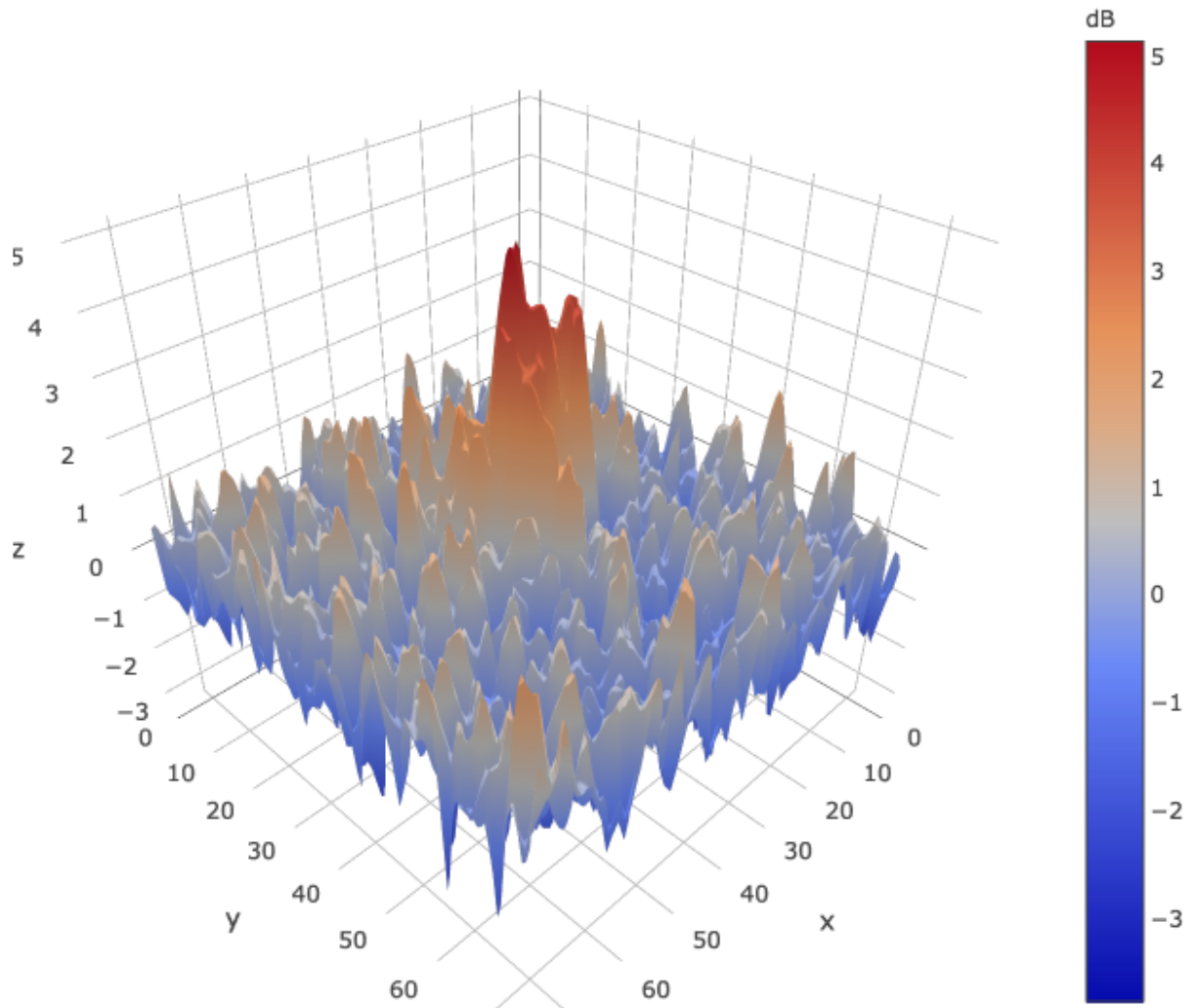
Our final submission has a private score of 0.2637 log loss! That is a noticeable improvement over our best baseline, the vanilla neural network's, score of 0.4553. Still good for middle-of-the-road in the Kaggle competition, but at least we're on the right track.

Close examiners of my Kaggle account (aka nobody besides me) will notice that a submission of [gpu\\_cnn.csv](#) had a lower log loss (0.24412) than the SimpNet architecture submission. Access to the supporting Jupyter notebook code was lost either through a git mishap or a Jupyter error, but rather than try to recreate something, I found a stable architecture in SimpNet that we can use as another baseline for improvement.

To that end, you might notice that even after normalizing, the decibel levels of the icebergs are still spikey, indicting elevated but not uniform backscatter.



iceberg\_f67babb0 | normalized dB



Bentes et al noted that "the strong backscattering represented by high pixel values can introduce instabilities in the training process. In addition to that, target's peak values do not generalize well in the classification task, [since] the high value is highly dependent of target orientation towards the satellite" (Bentes et al, 2016).

They therefore introduce a nonlinear normalization method,  $\mathbf{N}(\mathbf{x})$ , defined as:

$$N(x) = \frac{L(x)}{\max L(x)}$$

where

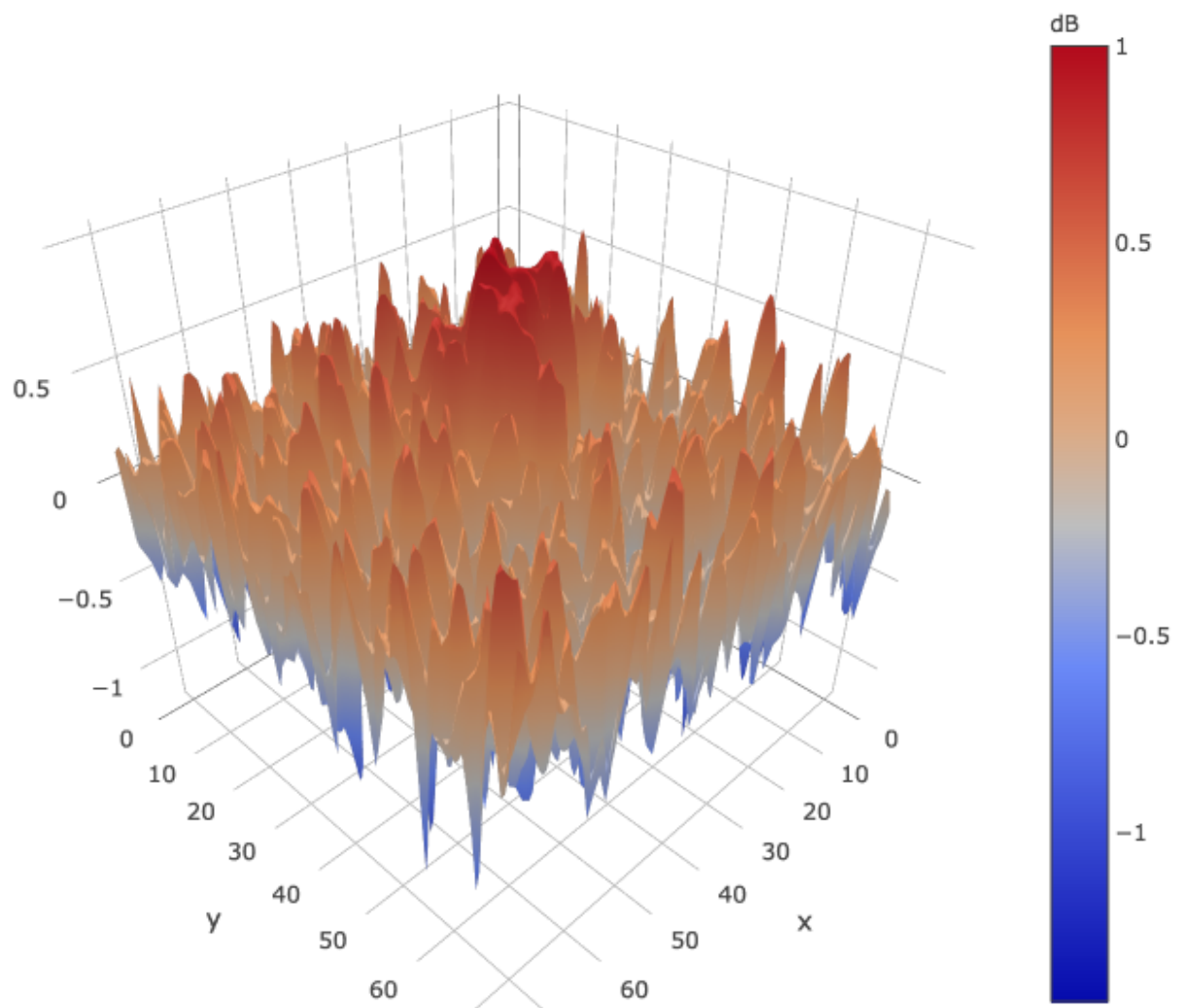
$$L(x) = \begin{cases} 1 + \log x & \text{if } x > 1 \\ x & \text{if } x \leq 1 \end{cases}$$

The L-function effectively blunts the higher decibel feedback that relates to a ship or an iceberg, since those are the highest decibels of backscatter in the image, to a closer range. As stated in the article, "the log



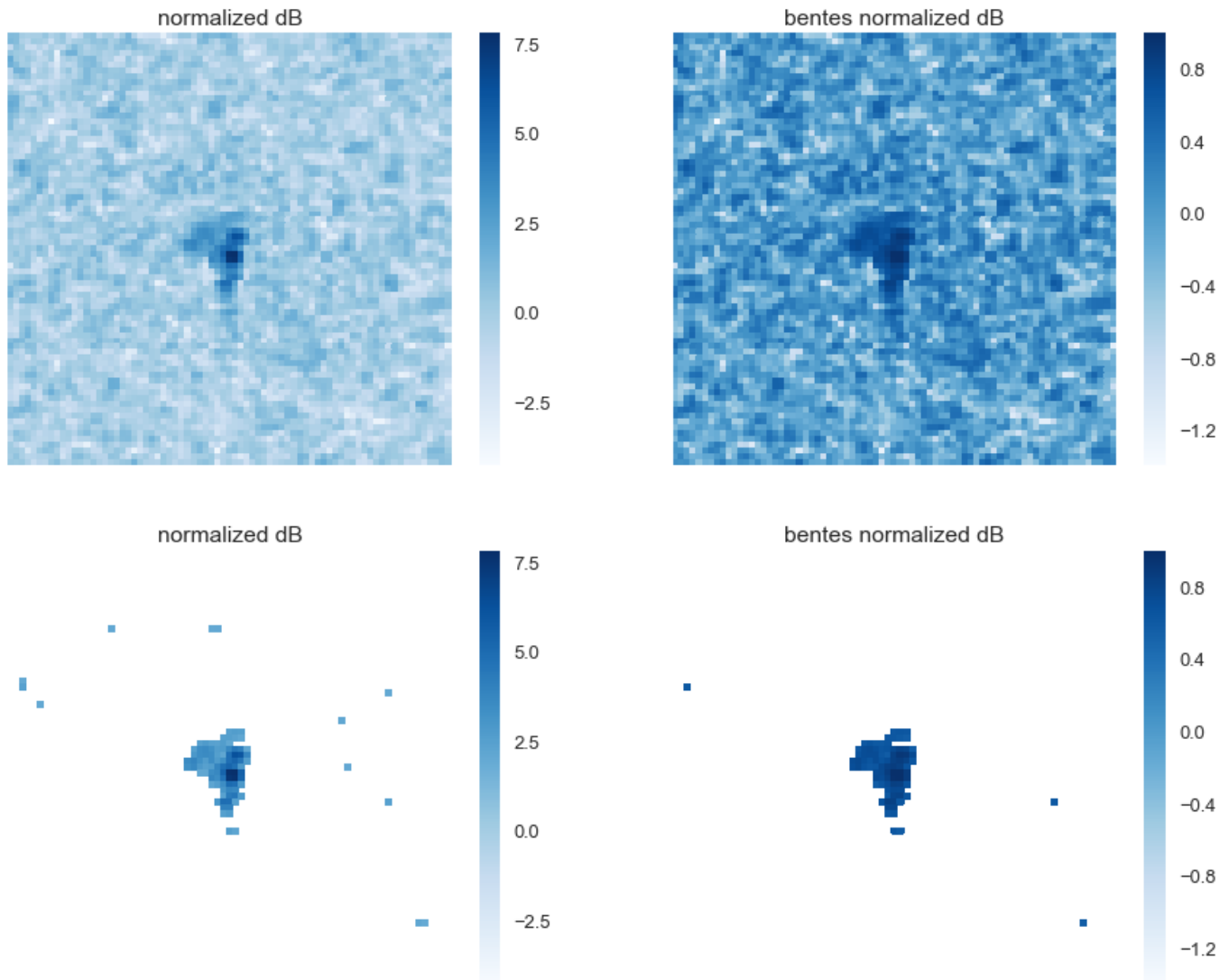
function...attenuates the high intensity values of the target signal" (Bentes et al, 2016).

iceberg\_f67babb0 | bentes normalized dB



It's a bit hard to tell from this static image, but the intensities of the ship or iceberg all lay around 1, while the background is muted to values between -0.5 and 0.5. Another image with an additional mask of 2 standard deviations above the mean is instructive:

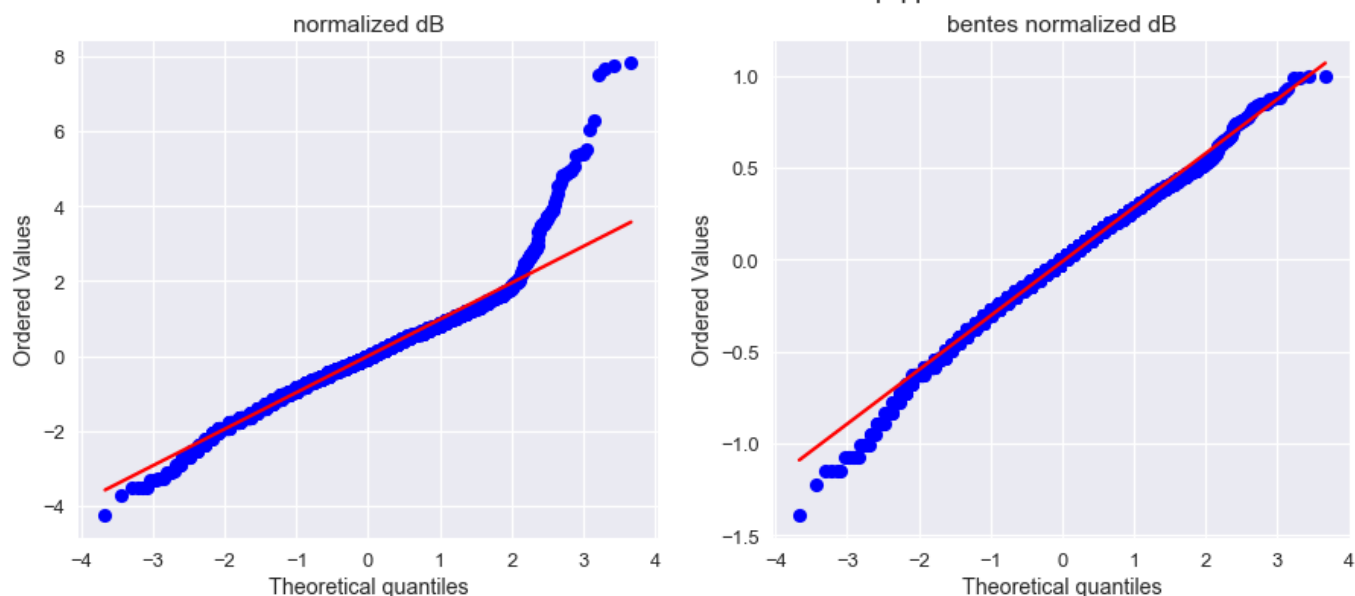
## iceberg\_17dcc03a with mask



Not only do less of the wave pixels get indicated with high pixel values, but the iceberg itself has a more uniform intensity, allowing our model to more easily find the edges where it changes to more background pixel values.

Furthermore, this "Bentes normalization", as I'm calling it, "changes the target pixel distribution from a highly skewed (right tail) distribution to a normal-like symmetric distribution" (Bentes et al, 2016). A before and after investigation into the Q-Q plot confirms this assertion.

effects of Bentes normalization on q-q plot

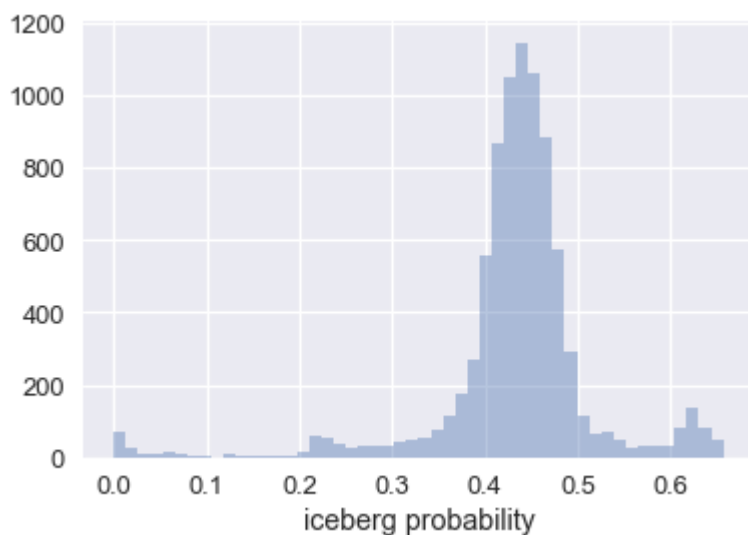


This particular iceberg had quite a satisfactory transformation to a normal-like spread.

Thus, with the new preprocessing steps enumerated in a new `make_tensors` function, we see if this new preprocessing has a positive effect on training a new SimpNet model with the same hyperparameters as before.

And from the early results, I can see that this approach failed at refinement. Other avenues involve improving or customizing the CNN architecture used for object recognition.

This is confirmed by the prediction distribution, but oddly this had the same final score as the SimpNet without Bentes normalization.



We prefer our original non-Bentes predictions because we were more confident in our guesses. Given the same log loss, we want a model that actually has predictive value.

## IV. Results

### Model Evaluation and Validation

Many trials of SimpNet and other architectures were attempted but not recorded for posterity. This was an oversight in record keeping and may not be a fixable error.

When the CNN architecture was custom-built, the number of filters, layer building, and percentage/location of dropouts were all tweaked.

The final model is reasonable and aligns with moderate solution expectations. The final parameters may not be optimal without building the best architecture from available layers, but are appropriate for a expired competition and an overdue capstone.

The model has been validated with held-back training data, but cross-fold validation was not used as requiring too much training time. The model has been tested on unseen testing data as provided by the competition.

The model is moderately robust. A way to test this would be to train it on non-augmented training data and see the change in final score. Another more direct way to measure robustness is to add Gaussian "noise" to the training data and monitor results, as a measure of atmospheric interference, but it would be of unknown value.

The last question of trusting the results is interesting. Certainly the log loss is low, but without further access to the labels of the testing data, it would be a guess to determine full trustworthiness. An accuracy on the training set of 89.6% is nothing to sneeze at, certainly. I would say if the model is confident, take it at face value. If the model is somewhat confident and an analyst can confirm, then the model may be used. Of course, obtaining more verified training examples will only serve to increase model accuracy, to a point.

Justification

We restate the improvement over the baseline log loss and assert that we have found a stronger result than the benchmark.

Baseline Model	is_iceberg =	Log Loss
certain iceberg	1.0	16.38164
certain not iceberg	0.0	18.15736
indecisive	0.5	0.6931
proportion of icebergs in training	0.469451	0.69808
Baseline Model	Log Loss	
Naive Neural Network	0.4553	
Final Model	Log Loss	
SimpNet; no Bentes	0.2637	

Since logarithmic loss is a measure of "closeness of prediction", and we obtained a probability of is\_iceberg distribution that made confident guesses on the testing set, then I argue the SimpNet result is a significant improvement over the naive neural network. I believe this model fully addresses the problem statement.

V. Conclusion

## Reflection

In all, we have preprocessed radar data, converted it to a 3-banded image that will be packaged into Keras tensors, and dealt with missing and outlying data, visualized the data in novel ways, built and trained a convolutional neural network to classify icebergs, and attempted (but failed) to improve on our techniques.

The interesting details and the ones I am most proud of in this project are mostly the visualization; it is there that I had to stretch the most. Building the CNN required knowledge and/or experimentation that I found difficult since there is no theory to rely on. I am disappointed that I did not document my steps to creating and training a custom CNN architecture; I have a vague memory that a shallower model with less augmentation performed better. If I had to pick one area to improve on it is the model creation and tweaking process. I need to monitor training with more visualizations and better record-keeping during training to apply deep learning in a more rigorous fashion. If I had to pick some code I'm most proud of, it's the custom augmentation generators that matched up augmented images with their incidence angles correctly.

Being that the model is a static copy of architecture from a different project domain, I believe that it is sub-optimal; or at least, not as good as I could personally do. This is evidenced by the lost code performing better than SimpNet. However, SimpNet serves the purpose of showing that a CNN performs the classification task better than our baseline models.

## Improvement

Beyond improving the model architecture itself, which has been discussed, since we have organized the SimpNet building process into blocks of layers, one could conceivably create a grid search of blocks and parameters to find a more optimal solution. This would drastically increase training time of course, but it is an avenue to solving the problem in a more thorough manner.

One way this model could be generalized is to make sure that it recognizes a 75x75 radar image of empty ocean as "not-iceberg" as well. At present, since our two classes both occupy high pixel values in the image, it is unknown what probability an image of only waves would return.

If we were to use my final solution as a new benchmark, then I have no doubt a better solution exists. If I submitted to Kaggle during the competition window, my model would land in the lower of the middle third of submissions. The leader has a log loss of about 0.08, so there must be techniques and improvements that I am not aware of.

However, I believe this is a good first foray into the world of machine learning. Hope you had as much anguished fun reading this report as I did writing it.